Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware

# Direct Programming with Intel® oneAPI DPC++/SYCL

Stephen Blair-Chappell
Intel oneAPI Certified Instructor

October 2024

# WHAT IS SYCL?

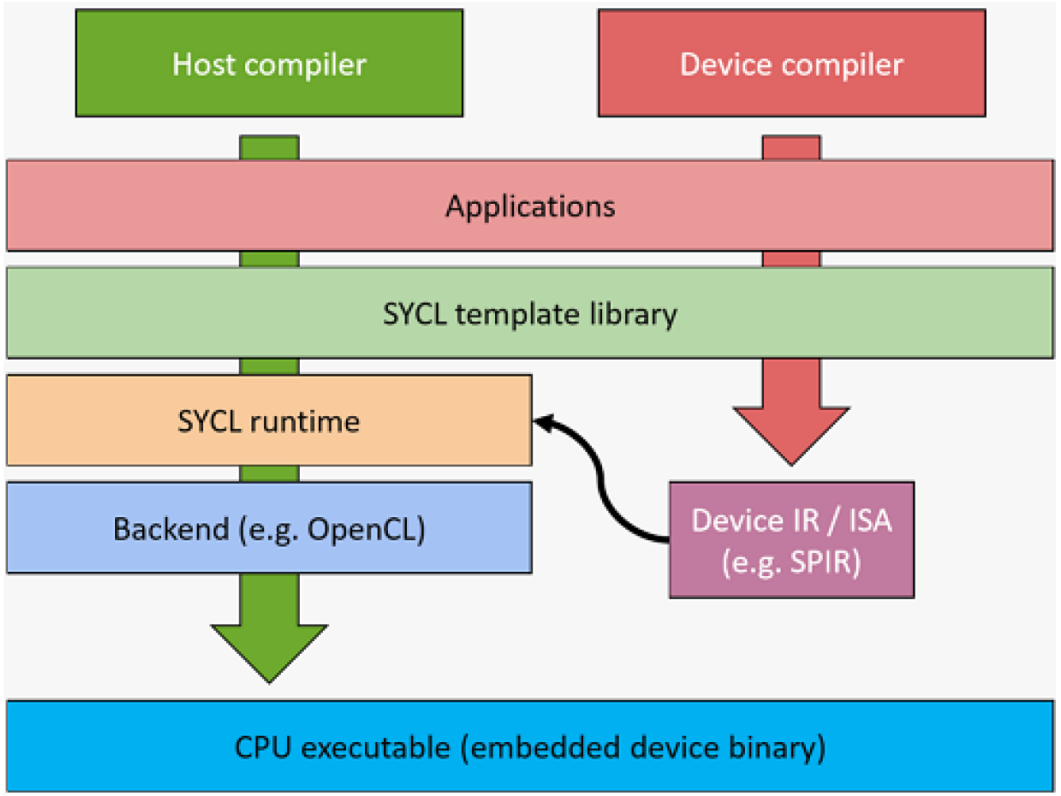
https://github.com/codeplaysoftware/syclacademy



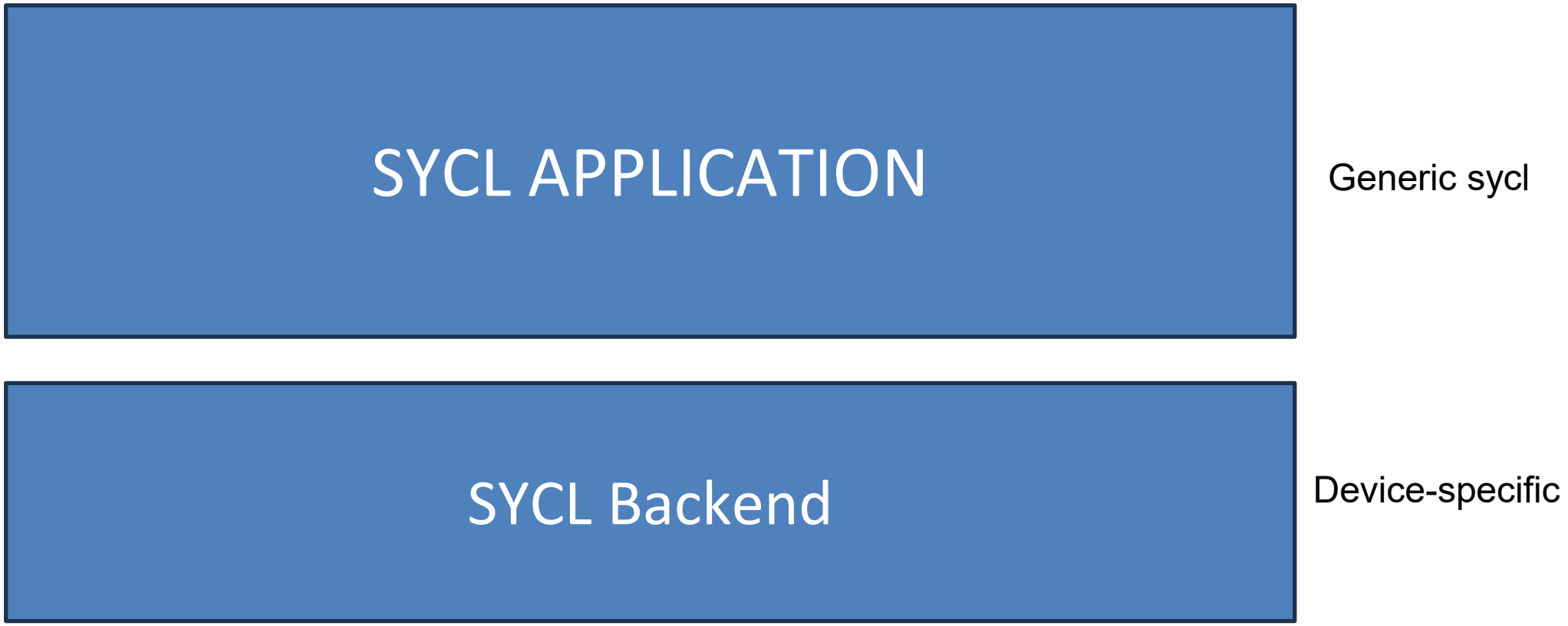
SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

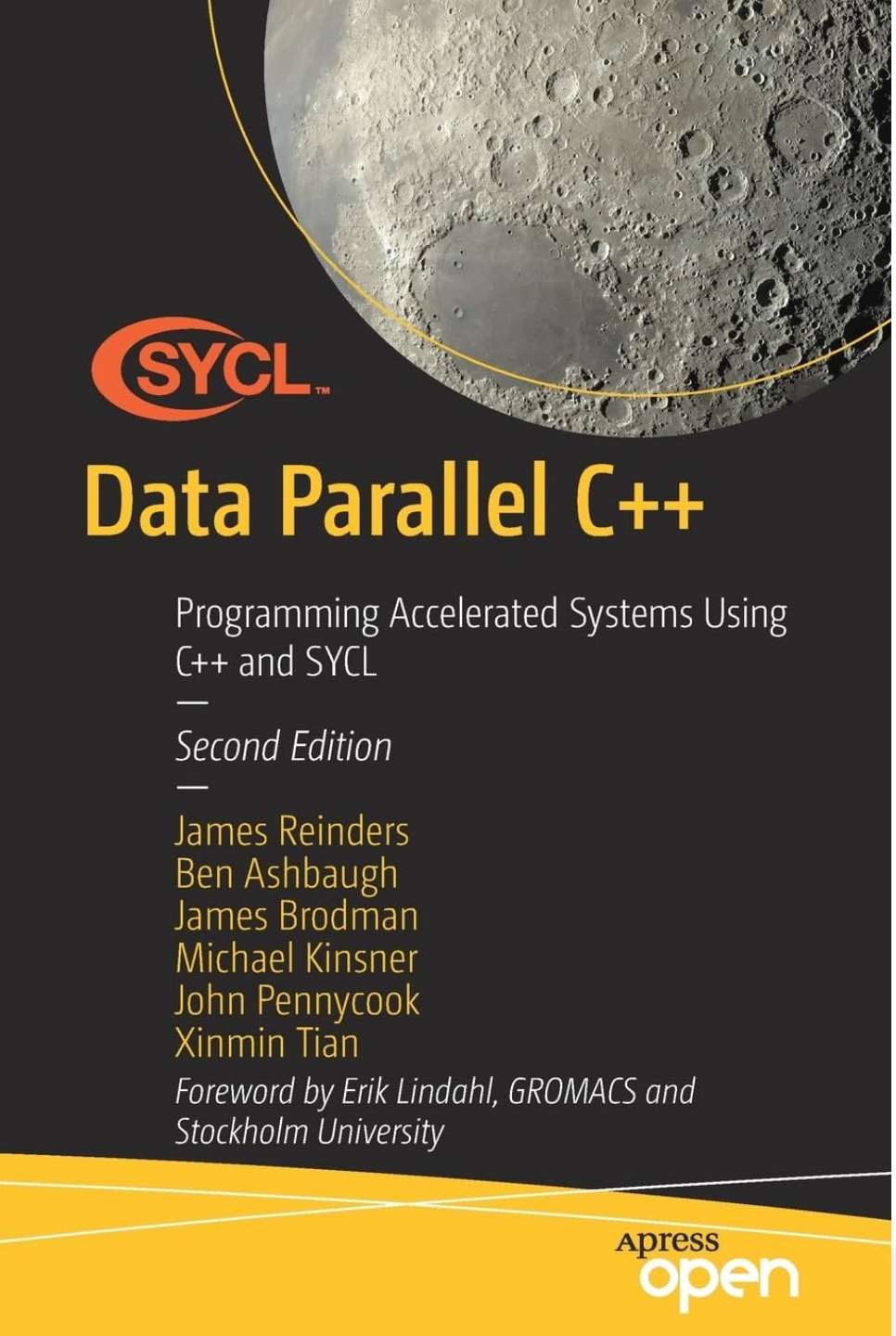
# WHAT IS SYCL?

SYCL is a **single source**, high-level, standard C++ programming model, that can target a range of heterogeneous platforms




- SYCL allows you to write both host CPU and device code in the same C++ source file
- This requires two compilation passes; one for the host code and one for the device code


SYCL APPLICATION
Generic sycl
SYCL Backend
Device-specific



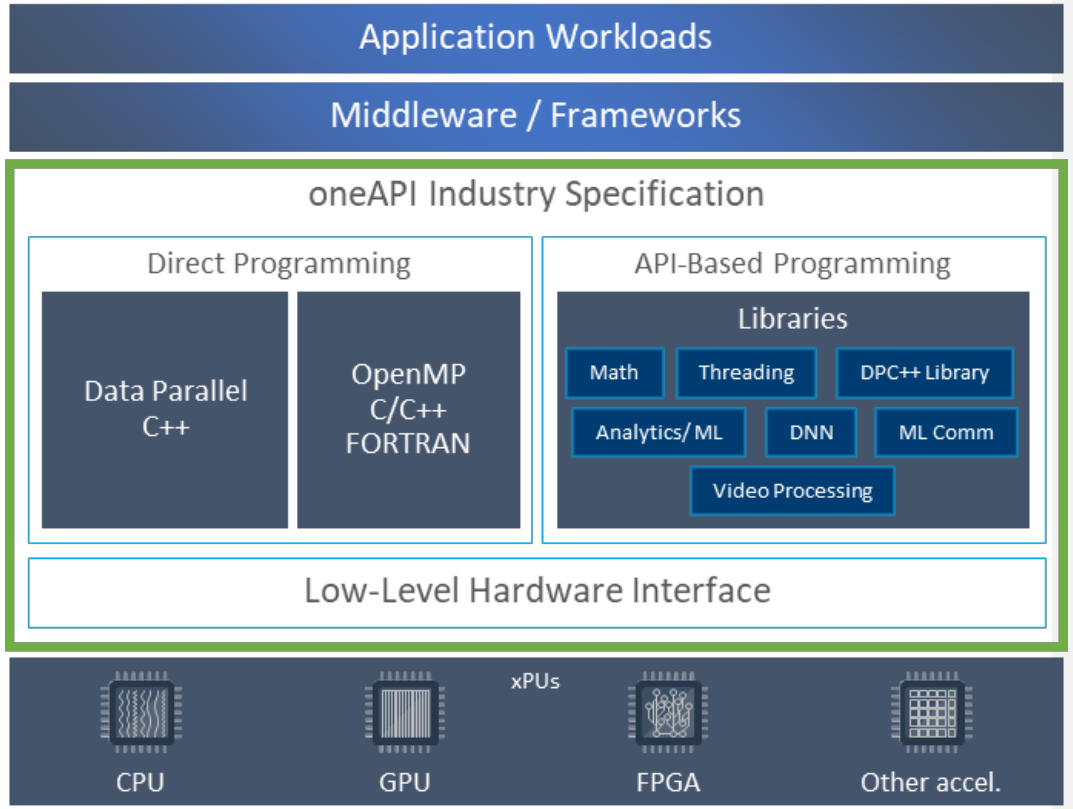

# Some Examples taken from this book

# Programmers' perspective: Three things to consider

1. Offload the code to the device

2. Manage the transfer of Data

3. Implement Parallelism

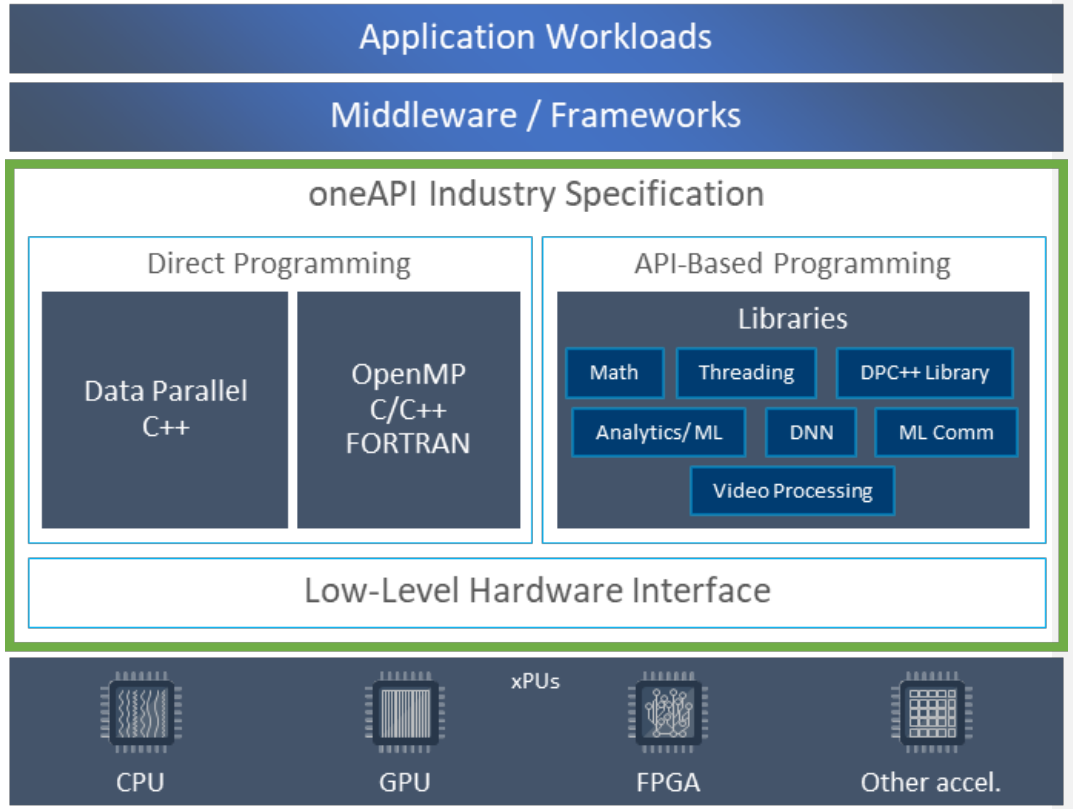


# Programmers' perspective: Three things to consider

1. Offload the code to the device

2. Manage the transfer of Data

3. Implement Parallelism



# Hello world (USM Example)

```
1.  #include <iostream>
2.  #include <sycl/sycl.hpp>
3.  using namespace sycl;
4.
5.  const std::string secret{
6.      "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
7.      "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01"};
8.
9.  const auto sz = secret.size();
10.
11. int main() {
12.   queue q;
13.
14.   char* result = malloc_shared<char>(sz, q);
15.   std::memcpy(result, secret.data(), sz);
16.
17.   q.parallel_for(sz, [=](auto& i) {
18.       result[i] -= 1;
19.    }).wait();
20.
21.   std::cout << result << "\n";
22.   free(result, q);
23.   return 0;
24. }
```

Host

Device

Host

**3** lets us avoid writing sycl:: over and over.

**12** instantiates a queue for work requests directed to a particular device .

**14** creates an allocation for data shared with the device

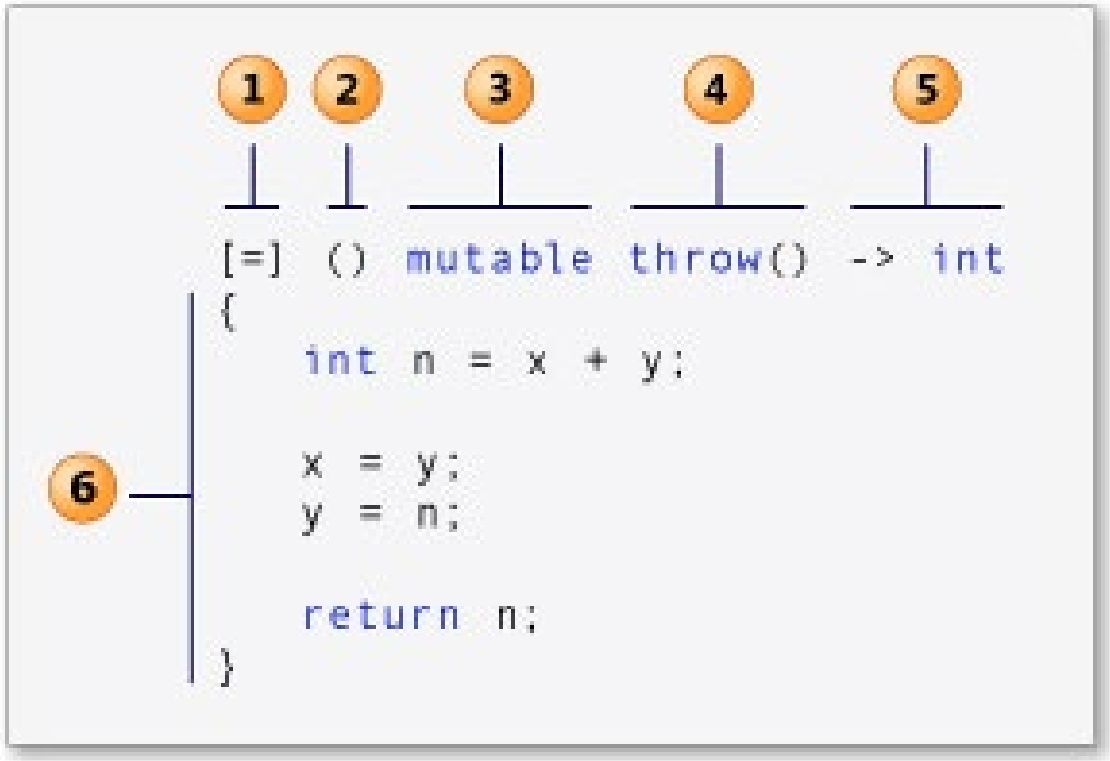
**15** copies the secret string into device memory, where it will be processed by the kernel.

**17** enqueues work to the device .

**18** is the only line of code that will run on the device. All other code runs on the host (CPU).

# A Reminder about Lambdas

```
16    Q.parallel_for(sz,[=](auto& i) {
17        result[i] -= 1;
18        }).wait();
```




*1.capture clause*

*2.parameter list* Optional.

*3.mutable specification* Optional.

*4.exception-specification* Optional.

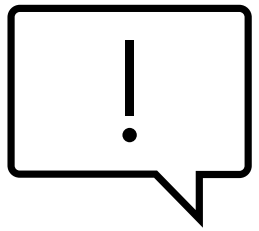
*5.trailing-return-type* Optional.

*6.lambda body.*

- [=] :capture by value
- [&]: capture by reference

https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp

# Kernel Code



```
16    Q.parallel_for(sz,[=](auto& i) {
17        result[i] -= 1;
18        }).wait();
```

```
./a.out
```

Kernel Code cannot use these features

- Run Asynchronously
- Limitation on what kind of C++ code
  - Dynamic Polymorphism
  - Dynamic memory allocations
  - Static variables
  - Function pointers
  - RTTI
  - Exception Handling
  - Recursion

# Hands-On Exercise 1 (5 minutes)

1) Use an editor to create code example(ex1.cpp)

2) Compile code

```
icpx -fsycl ex1.cpp -o ex1
```

3) Run program

```
./ex1
```

QUESTIONS:

- *Which lines of the code runs on the accelerator?*
- *Which Accelerator was used?*

```
#include <iostream>
#include <sycl/sycl.hpp>

int main() {
    try {
        // Step 1: Create a SYCL queue for the default accelerator
        sycl::queue q;

        // Print out the vendor and device name of the selected accelerator
        std::cout << "Running on: "
                  << q.get_device().get_info<sycl::info::device::vendor>()
                  << " "
                  << q.get_device().get_info<sycl::info::device::name>()
                  << std::endl;

        // Step 2: Offload "Hello World" using a lambda function in a kernel
        q.submit([&](sycl::handler &h) {
            // Create a SYCL stream object
            sycl::stream os(1024, 128, h);

            h.single_task([=]() {
                // Print "Hello World" from the accelerator using sycl::stream
                os << "Hello from SYCL!" << sycl::endl;
            });
        }).wait();
    } catch (sycl::exception const &e) {
        std::cout << "An exception occurred: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

# QUEUES & SELECTORS

# C++ SYCL for Offloading

## C++

```
#include <iostream>

int main(){



    // initialize some d
    const int N = 16;
    float data[N];
    for(int i=0;i<N;i++)

    // computation on CP

    for(int i=0;i<N;i++)


    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

1. Offload the code to the device
2. Manage the transfer of Data
3. Implement Parallelism

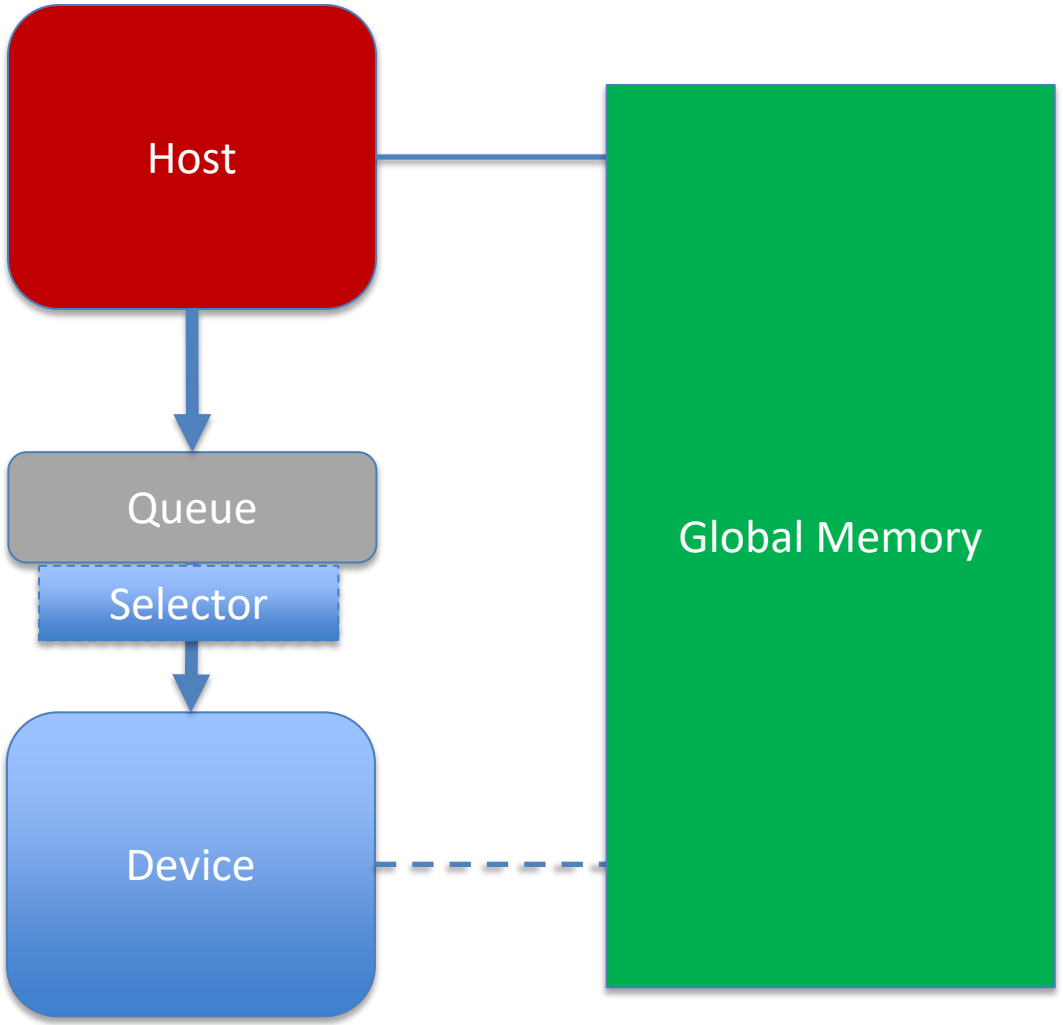
## C++ SYCL

```
#include <sycl/sycl.hpp>
#include <iostream>

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

Queue

Selector

# SYCL Queues




- *Queue* connects to a *device* via a *selector*
- Connection is made at *runtime* instantiation
- A queue *connection cannot change* once created
- More than one queue can point to the same device

# SYCL Basics - Selectors

| | |
|---|---|
| Get a device (any device): | `queue q (); // default_selector_v` |
| Create a queue with predefined device selectors | `queue q(cpu_selector_v);`<br>`queue q(gpu_selector_v);`<br>`queue q(accelerator_selector_v);` |
| Create a queue via custom selector | `int usm_selector(const sycl::device& dev) {`<br>`  if (dev.has(sycl::aspect::usm_device_allocations)) {`<br>`  if (dev.has(sycl::aspect::gpu)) return 2;`<br>`  return 1;`<br>`  }`<br>`  return -1;`<br>`}`<br>…<br>`queue q(usm_selector);` |

default_selector_v

- SYCL runtime scores all devices and picks one with highest compute power
- Environment variable

export ONEAPI_DEVICE_SELECTOR={backend:device_type:device_num}

# Choosing Devices : Six use cases:

| # | **Method** | **Comments** |
|---|---|---|
| 1 | Anywhere (don’t care where) | Runtime chooses |
| 2 | Always on Host CPU | Good for debugging |
| 3 | GPU or Accelerator | |
| 4 | Multiple devices | |
| 5 | Device with specific characteristics | e.g. ‘supports FP16’ |
| 6 | Custom (hand-coded selector) | |

# Method #1 (Execute Anywhere)

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // Create queue on whatever de                    ice that th
  // implementation choose                    plicit use of
  // default_selecto_v
  queue q;

  std::cout << "Selected device: "
            << q.get_device().get_info<info::device::name>()
            << "\n";

  return 0;
}
```

- Default Device used here
- Decided by the runtime
- No account taken of performance

Demo 01

# Queue class - Binding done at construction

```
class queue {
public:
  // Create a queue associated with the default device
  queue(const property_list = {});
  queue(const async_handler&,
        const property_list = {});

  // Create a queue associated with an explicit device
  // A device selector may be used in place of a device
  queue(const device&, const property_list = {});
  queue(const device&, const async_handler&,
        const property_list = {});

  // Create a queue associated with a device in a specific context
  // A device selector may be used in place of a device
  queue(const context&, const device&,
        const property_list = {});
  queue(const context&, const device&,
        const async_handler&,
        const property_list = {});
};
```

Default Device used here

Device/ Selector can passed in

# Runtime selection of devices ONEAPI_DEVICE_SELECTOR

| Example | Result |
|---|---|
| `ONEAPI_DEVICE_SELECTOR=opencl:*` | Only the OpenCL devices are available |
| `ONEAPI_DEVICE_SELECTOR=level_zero:gpu` | Only GPU devices on the Level Zero platform are available. |
| `ONEAPI_DEVICE_SELECTOR="opencl:gpu;level_zero:gpu"` | GPU devices from both Level Zero and OpenCL are available. Note that escaping (like quotation marks) will likely be needed when using semi-colon separated entries. |
| `ONEAPI_DEVICE_SELECTOR=opencl:gpu,cpu` | Only CPU and GPU devices on the OpenCL platform are available. |
| `ONEAPI_DEVICE_SELECTOR=opencl:0` | Only the device with index 0 on the OpenCL backend is available. |
| `ONEAPI_DEVICE_SELECTOR=hip:0,2` | Only devices with indices of 0 and 2 from the HIP backend are available. |
| `ONEAPI_DEVICE_SELECTOR=opencl:0.*` | All the sub-devices from the OpenCL device with index 0 are exposed as SYCL root devices. No other devices are available. |
| `ONEAPI_DEVICE_SELECTOR=opencl:0.2` | The third sub-device (2 in zero-based counting) of the OpenCL device with index 0 will be the sole device available. |
| `ONEAPI_DEVICE_SELECTOR=level_zero:*,*.*` | Exposes Level Zero devices to the application in two different ways. Each device (aka "card") is exposed as a SYCL root device and each sub-device is also exposed as a SYCL root device. |
| `ONEAPI_DEVICE_SELECTOR="opencl:*;!opencl:0"` | All OpenCL devices except for the device with index 0 are available. |
| `ONEAPI_DEVICE_SELECTOR="!*:cpu"` | All devices except for CPU devices are available. |

Benefits:

- Shared with OpenMP
- Allows selection of sub-devices

https://intel.github.io/llvm-docs/EnvironmentVariables.html#oneapi_device_selector

# Method #2 - Using CPU Device

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // Create queue to use the CPU device explicitly
  queue q{cpu_selector_v};

  std::cout << "Selected device: "
            << q.get_device().get_info<info::device::name>()
            << "\n";
  std::cout
      << " -> Device vendor: "
      << q.get_device().get_info<info::device::vendor>()
      << "\n";

  return 0;
}
```

- Q constructed with a CPU Selector
- Most debuggable – use standard tools (gcc, gprof etc)

Demo 02

# Method #3 – Using a GPU or Accelerator

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // Create queue bound to an available GPU device
  queue q{gpu_selector_v};

  std::cout << "Selected device: "
            << q.get_device().get_info<info::device::name>()
            << "\n";
  std::cout
      << " -> Device vendor: "
      << q.get_device().get_info<info::device::vendor>()
      << "\n";

  return 0;
}
```

- Q constructed with a GPU Selector
- If there are no GPUs available at runtime the selector will throw a runtime_error


(just change the selector to gpu_selector_v or accelerator_selector_v)

# Method #4 – Using Multiple Devices

```
#include <iostream>
#include <sycl/ext/intel/fpga_extensions.hpp>  // For fpga_selector_v
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  queue my_gpu_queue(gpu_selector_v);
  queue my_fpga_queue(ext::intel::fpga_selector_v);

  std::cout << "Selected device 1: "
            << my_gpu_queue.get_device()
                   .get_info<info::device::name>()
            << "\n";

  std::cout << "Selected device 2: "
            << my_fpga_queue.get_device()
                   .get_info<info::device::name>()
            << "\n";

  return 0;
}
```

- Two Queues
- Two Device Types

Demo 03

# Hands-On Exercise 2(10 minutes)

1) Use an editor to create code example(ex2.cpp)

2) Compile code

```
icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda
ex2.cpp -o ex2
```

3) Use the SYCL_PI_TRACE to confirm what accelerator the code ran on.

```
SYCL_PI_TRACE=1 ./ex2
```

QUESTIONS:

- *Which line of the code causes code to be associated with gpu?*
- *Can I change this behaviour at runtime with the* ONEAPI_DEVICE_SELECTOR *?*

```
ONEAPI_DEVICE_SELECTOR="*:cpu" ./ex2
```

- Why did we use the options -fsycl-targets= ?

```
#include <iostream>
#include <sycl/sycl.hpp>

int main() {
    try {
        // Step 1: Create a SYCL queue for gpu accelerator
        sycl::queue q(sycl::gpu_selector_v);

        // Print out the vendor and device name of the selected accelerator
        std::cout << "Running on: "
                  << q.get_device().get_info<sycl::info::device::vendor>()
                  << " "
                  << q.get_device().get_info<sycl::info::device::name>()
                  << std::endl;

        // Step 2: Offload "Hello World" using a lambda function in a kernel
        q.submit([&](sycl::handler &h) {
            // Create a SYCL stream object
            sycl::stream os(1024, 128, h);

            h.single_task([=]() {
                // Print "Hello World" from the accelerator using sycl::stream
                os << "Hello from SYCL!" << sycl::endl;
            });
        }).wait();
    } catch (sycl::exception const &e) {
        std::cout << "An exception occurred: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

# Method #5- Aspect Selector

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // In the aspect_selector form taking a comma seperated
  // group of aspects, all aspects must be present for a
  // device to be selected.
  queue q1{aspect_selector(aspect::fp16, aspect::gpu)};

  // In the aspect_selector form that takes two vectors, the
  // first vector contains aspects that a device must
  // exhibit, and the second contains aspects that must NOT
  // be exhibited.
  queue q2{aspect_selector(
      std::vector{aspect::fp64, aspect::fp16},
      std::vector{aspect::gpu, aspect::accelerator})};

  std::cout
      << "First selected device is: "
      << q1.get_device().get_info<info::device::name>()
      << "\n";

  std::cout
      << "Second selected device is: "
      << q2.get_device().get_info<info::device::name>()
      << "\n";

  return 0;
}
```

- **Example 1:** Device must support fp16 and be a GPU

- **Example 2:** Device must support fp16 and fp64 and must NOT be a gpu or other accelerator

# List of aspects available (in enum aspect::)

| Aspect | |
|---|---|
| cpu | online_compiler |
| gpu | online_linker |
| Accelerator | queue_profiling |
| custom | usm_device_allocations |
| emulated | usm_host_allocations |
| host_debuggable | usm_atomic_host_allocations |
| fp16 | usm_shared_allocations |
| fp64 | usm_atomic_shared_allocations |
| atomic64 | usm_system_allocations |
| image | |

# Method #6- Custom Selector

```
int my_selector(const device &dev) {
  if (dev.get_info<info::device::name>().find("pac_a10") !=
          std::string::npos &&
      dev.get_info<info::device::vendor>().find("Intel") !=
          std::string::npos) {
    return 1;
  }
  return -1;
}
```

- Define the custom selector

- Return a positive value if to be used.
- Highest value selector ‘wins’

**Example use : see next slide**

```
auto q = queue{my_selector};
```

Demo 04

# Method #6- Custom Selector

```
auto GPU_is_available = false;

try {
  device testForGPU(my_selector);
  GPU_is_available = true;
} catch (exception const& ex) {
  std::cout << "Caught this SYCL exception: " << ex.what()
            << std::endl;
}

auto q = GPU_is_available ? queue(my_selector)
                          : queue(default_selector_v);
```

- Test if Device is available using custom selector

- If available use custom selector, otherwise use default selector

Demo 04

# Runtime Default Selector Scoring

https://github.com/intel/llvm/blob/sycl/sycl/source/device_selector.cpp

```
183
184     if (dev.is_gpu())
185       Score += 500;
186
187     if (dev.is_cpu())
188       Score += 300;
189
190     // Since we deprecate SYCL_BE and SYCL_DEVICE_TYPE,
191     // we should not disallow accelerator to be chosen.
192     // But this device type gets the lowest heuristic point.
193     if (dev.is_accelerator())
194       Score += 75;
195
196     // Add preference score.
197     Score += detail::getDevicePreference(dev);
198
199     return Score;
```

```
31    // ONEAPI_DEVICE_SELECTOR doesn't need to be considered in the device
32    // preferences as it filters the device list returned by device::get_devices
33    // itself, so only matching devices will be scored.
34    static int getDevicePreference(const device &Device) {
35      int Score = 0;
36
37      // Strongly prefer devices with available images.
38      auto &program_manager = sycl::detail::ProgramManager::getInstance();
39      if (program_manager.hasCompatibleImage(Device))
40        Score += 1000;
41
42      // Prefer level_zero backend devices.
43      if (detail::getSyclObjImpl(Device)->getBackend() ==
44          backend::ext_oneapi_level_zero)
45        Score += 50;
46
47      return Score;
48    }
```

# Code to find scores

```
1   #include <sycl/sycl.hpp>
2   #include <iostream>
3
4   int main() {
5     // Get all available devices
6     auto devices = sycl::device::get_devices();
7
8     for (const auto &dev : devices) {
9       // Use the callable default_selector_v to get the score
10      int score = sycl::default_selector_v(dev);
11      std::cout << "Device: " << dev.get_info<sycl::info::device::name>()
12                << " Score: " << score << std::endl;
13    }
14    return 0;
15  }
```

```
stephen@HP-OMEN-017:~/dv/BIRMINGHAM/tmp$ ./a.out
Device: 13th Gen Intel(R) Core(TM) i9-13900HX Score: 300
Device: Intel(R) Graphics [0xa788] Score: 500
Device: Intel(R) Graphics [0xa788] Score: 550
Device: NVIDIA GeForce RTX 4090 Laptop GPU Score: 500
```

# Hands-On Exercise 3(10 minutes)

1) Use an editor to create code example(ex3.cpp)

2) Compile code

```
icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda
ex3.cpp -o ex3
```

3) Use the SYCL_PI_TRACE to confirm what accelerator the code ran on.

```
SYCL_PI_TRACE=1 ./ex3
```

QUESTIONS:

- *Which line of the code causes code to be associated with gpu?*
- *Can I change this behaviour at runtime with the* ONEAPI_DEVICE_SELECTOR *?*

```
ONEAPI_DEVICE_SELECTOR="*:cpu" ./ex3
```

- What happens if you don't use the options -fsycl-targets= ?
- How might I make sure that code running on my laptop gives priority of NVIDIA GPU over Intel GPU

```
[opencl:cpu][opencl:0] Intel(R) OpenCL, 13th Gen Intel(R) Core(TM) i9-13900HX OpenCL 3.0 (Build
0) [2024.18.6.0.02_160000]
[opencl:gpu][opencl:1] Intel(R) OpenCL Graphics, Intel(R) Graphics [0xa788] OpenCL 3.0 NEO
[23.43.27642.52]
[level_zero:gpu][level_zero:0] Intel(R) Level-Zero, Intel(R) Graphics [0xa788] 1.3 [1.3.27642]~
[cuda:gpu][cuda:0] NVIDIA CUDA BACKEND, NVIDIA GeForce RTX 4090 Laptop GPU 8.9 [CUDA 12.5]
```

```
#include <sycl/sycl.hpp>
#include <iostream>

// Define a custom device selection function
int custom_device_selector(const sycl::device& dev) {
    // Prioritize GPUs
    if (dev.is_gpu()) {
        return 1000;  // High score for GPUs
    }
    return -1;  // Ignore non-GPU devices
}

int main() {
    // Create a SYCL queue using the custom device selector function
    sycl::queue q(custom_device_selector);

    // Print the name of the selected device
    std::cout << "Running on: "
              << q.get_device().get_info<sycl::info::device::name>()
              << std::endl;

    return 0;
}
```

# Queue selection - Programming approaches

- Just let the runtime make the decision for you (most portable) – use default queue
- Hard code accelerator in code (type, aspect or custom)
- Use Runtime environment variable

  ONEAPI_DEVICE_SELECTOR

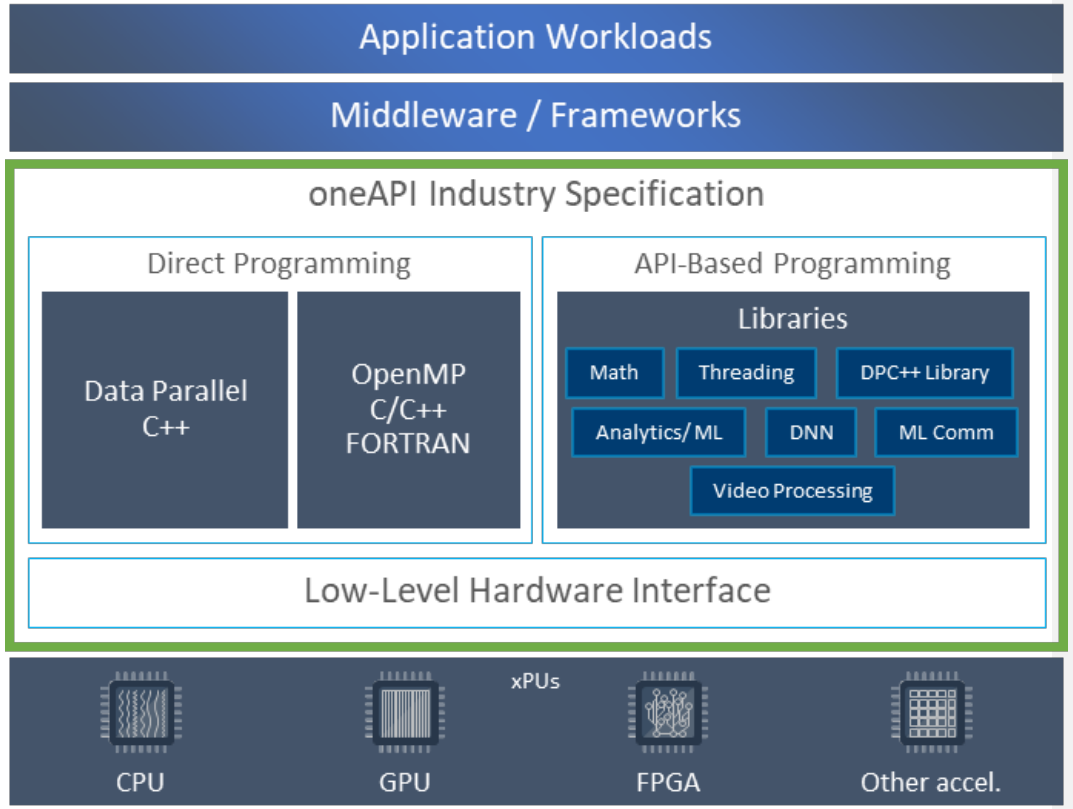
  *e.g. the Intel Developer Cloud uses ONEAPI_DEVICE_SELECTOR to give the user just ONE gpu when using Jupyter Notebooks*

# Programmers' perspective: Three things to consider

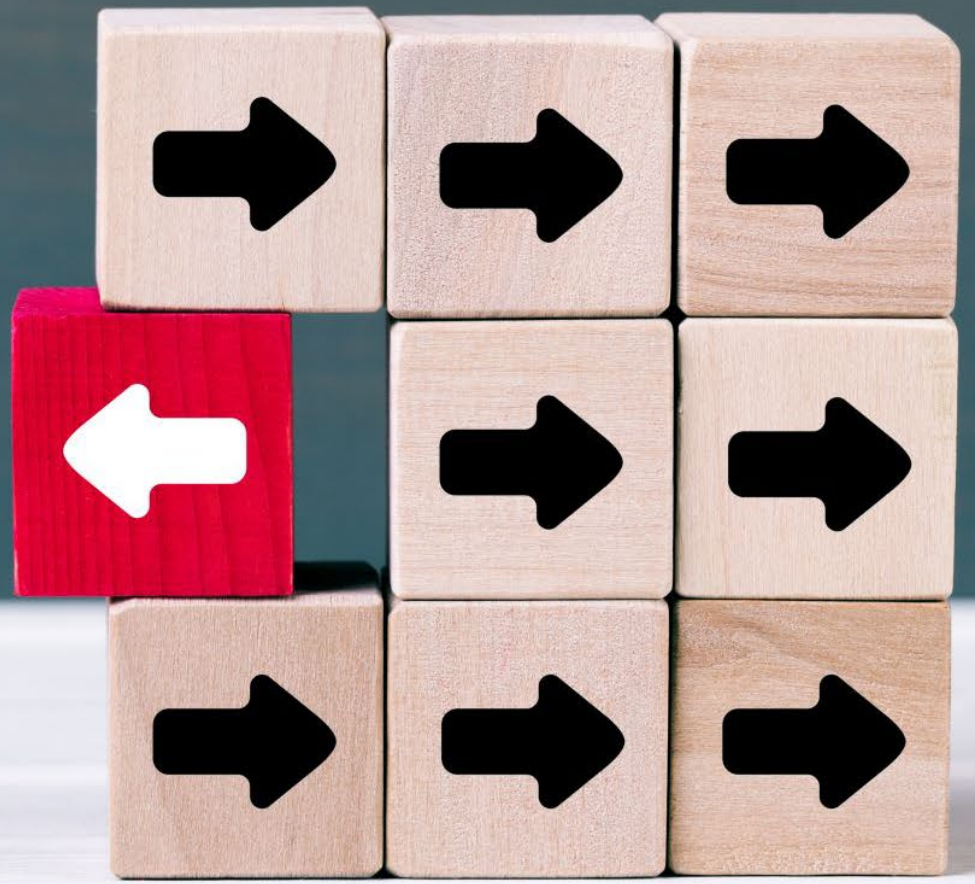
1. Offload the code to the device

2. Manage the transfer of Data

3. Implement Parallelism



# Time to Choose
# Buffers and Accessors or USM



# SYCL Timeline

**Memory Model**

2014: Initial release of SYCL
2015: SYCL 1.2 final version
2016: SYCL 2.2 provisional version
2017: SYCL 1.2.1 first version
2020: SYCL 1.2.1 revision 7

Buffers & Accessors

2020: SYCL 2020 Provisional Specification
- USM added here

2021: SYCL 2020 Specification
2022: SYCL 2020 revision 6
2023: SYCL 2020 revision 8

# USM vs Buffers/Accessors

## USM Example

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main(int, char**) {
    const int size = 10000;
    queue q;
    // USM allocation, implicit data movement
    float* x = malloc_shared<float>(size, q);
    float* y = malloc_shared<float>(size, q);
    // … initialization of x_vec, y_vec and a

    range<1> num_items{ size };
    q.submit([&](handler& h) {
        h.parallel_for(num_items, [=](item<1> i) {
            y[i] = a * x[i] + y[i];
        });
    });
    q.wait();
    // … print results
    free(x, q);
    free(y, q);
    return 0;
}
```

## Buffers/Accessors Example

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main(int, char**) {
  const int size = 10000;
  std::vector<float> x_vec(size, 1.0f);
  std::vector<float> y_vec(size, 2.0f);
  float a = 0.5;

  queue q;
  buffer x_buf(x_vec);
  buffer y_buf(y_vec);
  range<1> num_items{ x_vec.size() };
  q.submit([&](handler& h) {
    accessor x(x_buf, h, read_only);
    accessor y(y_buf, h, read_write);
    h.parallel_for(num_items, [=](item<1> i) {
      y[i] = a * x[i] + y[i];
    });
  });
  host_accessor y_res(y_buf, read_only);
  // … print results and returns
}
```

# TWO Kinds of programming using SYCL

- **USM (Unified Shared Memory).**
  Lower-level pointer-based solution with fine grained control.
- **buffer/accessors**
  Abstract model of memory provides guaranteed consistency and automatically manages dependencies.
- **image/accessors**

- CUDA migration.
- C++ pointers
- Porting existing applications

- Prototyping,
- Provides info to runtime that helps optimisation.
- Max performance portability required

- Image manipulation

# Hello world (USM Example) – example from earlier

```
1.  #include <iostream>
2.  #include <sycl/sycl.hpp>
3.  using namespace sycl;
4.
5.  const std::string secret{
6.      "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
7.      "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01"};
8.
9.  const auto sz = secret.size();
10.
11. int main() {
12.   queue q;
13.
14.   char* result = malloc_shared<char>(sz, q);
15.   std::memcpy(result, secret.data(), sz);
16.
17.   q.parallel_for(sz, [=](auto& i) {
18.       result[i] -= 1;
19.    }).wait();
20.
21.   std::cout << result << "\n";
22.   free(result, q);
23.   return 0;
24. }
```

Host

Device

Host

**3** lets us avoid writing sycl:: over and over.

**12** instantiates a queue for work requests directed to a particular device .

**14** creates an allocation for data shared with the device

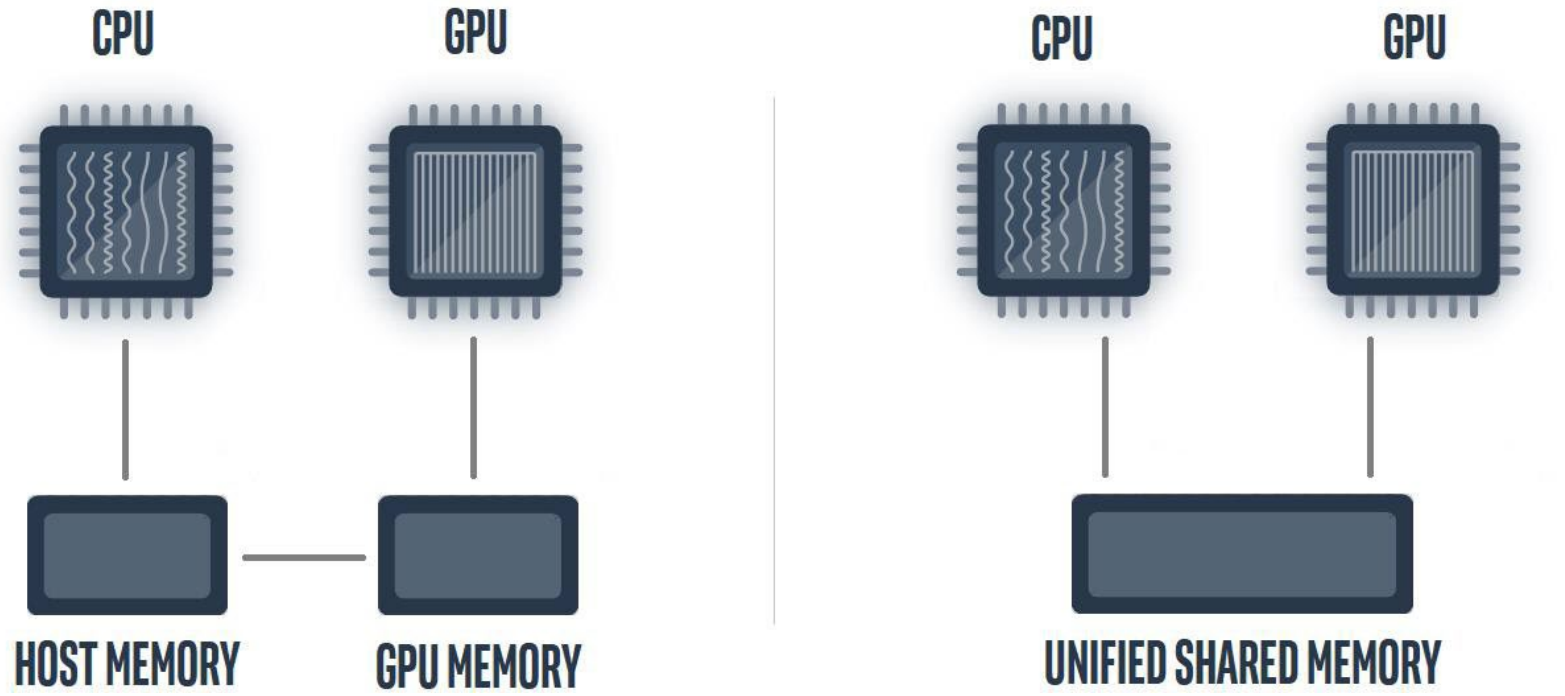
**15** copies the secret string into device memory, where it will be processed by the kernel.

**17** enqueues work to the device .

**18** is the only line of code that will run on the device. All other code runs on the host (CPU).

# Developer View Of USM

- Developers can reference same memory object in host and device code with Unified Shared Memory



# Dynamic memory allocation

| *Allocation Functions* | *Allocation Type* |
|---|---|
| `malloc_device(…)` | device |
| `aligned_alloc_device(…)` | device |
| `malloc_host(…)` | host |
| `aligned_alloc_host(…)` | host |
| `malloc_shared(…)` | shared |
| `aligned_alloc_shared(…)` | shared |
| `free(ptr,q)` | deallocate |

See: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_usm_allocations

# Characteristics of USM allocations

| Allocation Type | Initial Location | Accessible By | | Migratable To | |
|---|---|---|---|---|---|
| `device` | `device` | `host` | No | `host` | No |
| | | `device` | Yes | `device` | N/A |
| | | Another `device` | Optional (P2P) | Another `device` | No |
| `host` | `host` | `host` | Yes | `host` | N/A |
| | | Any `device` | Yes | `device` | No |
| `shared` | Unspecified | `host` | Yes | `host` | Yes |
| | | `device` | Yes | `device` | Yes |
| | | Another `device` | Optional | Another `device` | Optional |

See: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_usm_allocations

# Characteristics of USM allocations

| Allocation Type | Initial Location | Accessible By | | Migratable To | |
|---|---|---|---|---|---|
| `device` | `device` | `host` | No | `host` | No |
| | | `device` | Yes | `device` | N/A |
| | | Another `device` | Optional (P2P) | Another `device` | No |
| `host` | `host` | `host` | Yes | `host` | N/A |
| | | Any `device` | Yes | `device` | No |
| `shared` | Unspecified | `host` | Yes | `host` | Yes |
| | | `device` | Yes | `device` | Yes |
| | | Another `device` | Optional | Another `device` | Optional |

Host and shared allocations accessible from anywhere

# Characteristics of USM allocations

| Allocation Type | Initial Location | Accessible By | | Migratable To | |
|---|---|---|---|---|---|
| `device` | `device` | `host` | No | `host` | No |
| | | `device` | Yes | `device` | N/A |
| | | Another `device` | Optional (P2P) | Another `device` | No |
| `host` | `host` | `host` | Yes | `host` | N/A |
| | | Any `device` | Yes | `device` | No |
| `shared` | Unspecified | `host` | Yes | `host` | Yes |
| | | `device` | Yes | `device` | Yes |
| | | Another `device` | Optional | Another `device` | Optional |

All allocation types can be accessed from another device

# Characteristics of USM allocations

| Allocation Type | Initial Location | Accessible By | | Migratable To | |
|---|---|---|---|---|---|
| `device` | `device` | `host` | No | `host` | No |
| | | `device` | Yes | `device` | N/A |
| | | Another `device` | Optional (P2P) | Another `device` | No |
| `host` | `host` | `host` | Yes | `host` | N/A |
| | | Any `device` | Yes | `device` | No |
| `shared` | Unspecified | `host` | Yes | `host` | Yes |
| | | `device` | Yes | `device` | Yes |
| | | Another `device` | Optional | Another `device` | Optional |

Only shared allocation types will migrate – i.e. implicitly moved by the runtime

# Data Movement – Two Types

- Implicit - taken care of by runtime
  - Shared allocations implicitly share data between the host and devices. Data may move to where it is being used without the programmer explicitly informing the runtime. It is up to the runtime and backends to make sure that a shared allocation is available where it is used. Shared allocations must also be obtained using SYCL allocation routines instead of the system allocator. The maximum size of a shared allocation on a specific device, and the total size of all shared allocations in a context, are implementation-defined. Support for shared allocations on a specific device can be queried through `aspect::usm_shared_allocations`.
- Explicit – under programmer control
  - Device allocations are used for explicitly managing device memory. Programmers directly allocate device memory and explicitly copy data between host memory and a device allocation. Device allocations are obtained through SYCL device USM allocation routines instead of system allocation routines like `std::malloc` or C++ `new`. Device allocations are not accessible on the host, but the pointer values remain consistent on account of Unified Addressing. The size of device allocations will be limited by the amount of memory in a device. Support for device allocations on a specific device can be queried through `aspect::usm_device_allocations`.

# USM - Implicit Data Movement

```
queue q;
int *hostArray = (int*) malloc_host(42 * sizeof(int), q);
int *sharedArray = (int*) malloc_shared(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 1234;
q.submit([&](handler& h){
  h.parallel_for(42, [=](auto ID) {
    // access sharedArray and hostArray on device
    sharedArray[ID] = hostArray[ID] + 1;
  });
});
q.wait();
for (int i = 0; i < 42; i++) hostArray[i] = sharedArray[i];
free(sharedArray, q);
free(hostArray, q);
```

# USM - Implicit Data Movement

```
queue q;
int *hostArray = (int*) malloc_host(42 * sizeof(int), q);
int *sharedArray = (int*) malloc_shared(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 1234;
q.parallel_for(42, [=](auto ID) {

    // access sharedArray and hostArray on device
    sharedArray[ID] = hostArray[ID] + 1;
  });
});
q.wait();
for (int i = 0; i < 42; i++) hostArray[i] = sharedArray[i];
free(sharedArray, q);
free(hostArray, q);
```

# USM - Explicit Data Movement

```
queue q;
int hostArray[42];
int *deviceArray = (int*) malloc_device(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 42;
// copy hostArray to deviceArray
q.memcpy(deviceArray, &hostArray[0], 42 * sizeof(int));
q.wait();
q.submit([&](handler& h){
  h.parallel_for(42, [=](auto ID) {
    deviceArray[ID]++;
  });
});
q.wait();
// copy deviceArray back to hostArray
q.memcpy(&hostArray[0], deviceArray, 42 * sizeof(int));
q.wait();
free(deviceArray, q);
```

# USM - Explicit Data Movement

```
queue q;
int hostArray[42];
int *deviceArray = (int*) malloc_device(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 42;
// copy hostArray to deviceArray
q.memcpy(deviceArray, &hostArray[0], 42 * sizeof(int));
q.wait();
q.parallel_for(42, [=](auto ID) {

    deviceArray[ID]++;
  });
});
q.wait();
// copy deviceArray back to hostArray
q.memcpy(&hostArray[0], deviceArray, 42 * sizeof(int));
q.wait();
free(deviceArray, q);
```

# Common SYCL_PI_TRACE Output Elements: 1 of 3

1. **Platform Information:** When you set `SYCL_PI_TRACE=2`, the output typically starts by displaying detailed information about the platforms and devices that the SYCL runtime detects. You might see entries like:

```lua
--> piPlatformsGet
<-- Result: PI_SUCCESS
```

   This indicates that the runtime is querying the platforms, and `PI_SUCCESS` confirms that it found one or more platforms successfully.

2. **Device Querying:** Following the platform detection, you'll see calls related to device detection:

```lua
--> piDevicesGet
<-- Result: PI_SUCCESS
```

   This traces the call to query devices for each platform. The devices (e.g., CPUs, GPUs, FPGAs) available to SYCL are detected and listed here.

# Common SYCL_PI_TRACE Output Elements: 2 of 3

3. **Memory Allocations:** When your SYCL program uses Unified Shared Memory (USM), the trace output shows the allocation of memory:

```lua
--> piMemBufferCreate
<-- Result: PI_SUCCESS
```

This indicates that memory is being allocated, and the runtime confirms that it was successful.

4. **Kernel Submission:** Each time you submit a kernel (e.g., `parallel_for`), `SYCL_PI_TRACE` shows the sequence of events:

```diff
--> piEnqueueKernelLaunch
```

This traces the kernel launch, indicating that the kernel is being offloaded to the selected device. Following the launch, you will see a series of trace messages indicating synchronization points or memory access:

```lua
--> piEventsWait
<-- Result: PI_SUCCESS
```

# Common SYCL_PI_TRACE Output Elements: 3 of 3

5. **Memory Transfers:** If there is any data transfer between the host and the device (or between different devices), the trace logs these operations. For instance, in USM, if there is implicit memory copying happening under the hood, you might see:

```lua
--> piEnqueueMemBufferRead
<-- Result: PI_SUCCESS
```

   This indicates that data is being read from the device memory to the host, or vice versa, depending on the operation.

6. **Synchronization:** When SYCL waits for an operation to complete (e.g., calling `q.wait()`), the output logs the event synchronization:

```lua
--> piEventsWait
<-- Result: PI_SUCCESS
```

   This indicates that the runtime is waiting for the queued kernel to complete its execution on the device.

# Hands-On Exercise 4 & 5 (20 minutes)

1) Use an editor to create code example(**ex4.cpp**) and fill in the missing code

2) Compile code

```
icpx -fsycl ex4.cpp -o ex4
```

3) Use the SYCL_PI_TRACE to confirm what memory allocations and memory transfers take place

```
SYCL_PI_TRACE=2 ./ex4
```

4) Save a copy of ex4.cpp as **ex5.cpp** and change the code so it uses EXPLICIT memory transfer rather than IMPLICIT memory transfer

5) Compile code

```
icpx -fsycl ex5.cpp
```

6) Use the SYCL_PI_TRACE to confirm what memory allocations and memory transfers take place

```
SYCL_PI_TRACE=2 ./ex5
```

QUESTIONS:

- *Explain the difference between the two code examples?*
- Is the trace different between the two examples – can you explain why?

```
#include <iostream>
#include <sycl/sycl.hpp>

int main() {
    // Step 1: Create a SYCL queue
    // TODO                          ;

    // Step 2: Allocate shared memory using USM
    const int N = 16;
    int *data =                                ;

    // Step 3: Initialize the array on the host
    for (int i = 0; i < N; i++) {
        data[i] = i;
    }

    // Step 4: Submit a kernel to the device to add 5 to each element
    q.parallel_for(N, [=](sycl::id<1> i) {
        /* Add 5 to each element of 'data' */
        //TODO                          ;
    }).wait();

    // Step 5: Print the updated values on the host
    for (int i = 0; i < N; i++) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }

    // Step 6: Free the allocated memory
    // TODO                          ;

    return 0;
}
```

# USM - Data Dependency in Queues

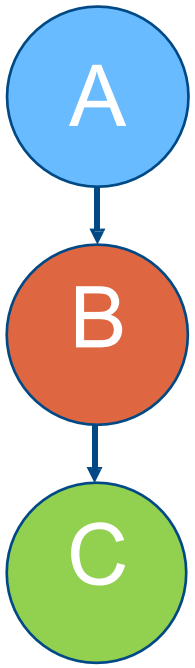
Dependences must be specified explicitly using events

- queue.wait()
- wait on event objects
- use the depends_on method inside a command group

# USM - Data Dependency in Queues

Explicit wait() used to ensure data dependency in maintained

wait() will block execution on host

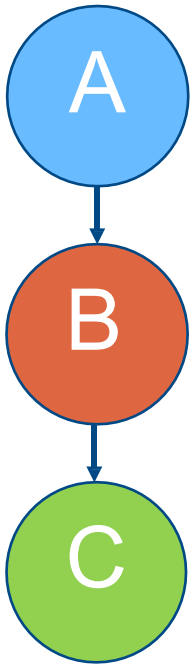



```
queue q;
int* data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
}).wait();
q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data[i] += 3;
    });
}).wait();
q.submit([&] (handler &h){
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

# USM - Data Dependency in Queues

## Use in_order property for the queue

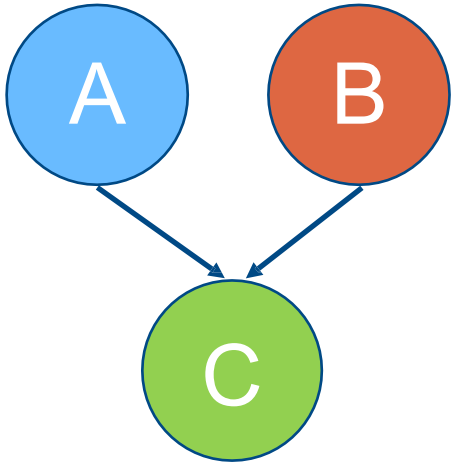
Execution will not overlap even if the queues have no data dependency




```
queue q{property::queue::in_order()};
int *data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
});
// non-blocking; execution of host code is possible
q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data[i] += 3;
    });
});
// non-blocking; execution of host code is possible
q.submit([&] (handler &h){
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

# USM - Data Dependency in Queues

Use depends_on() method to let command group handler know that specified events should be complete before specified task can execute

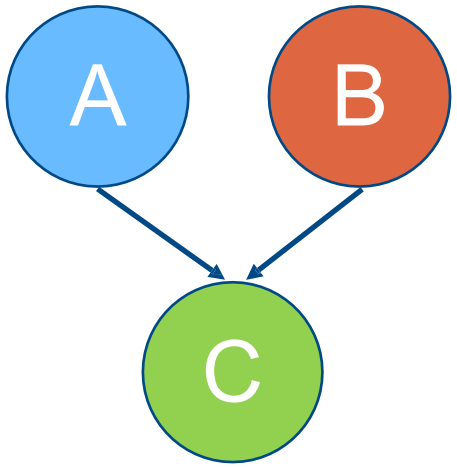



```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data1[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data2[i] += 3;
    });
});
q.submit([&] (handler &h){
    h.depends_on({e1,e2});
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data1[i] += data2[i];
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data1, q); free(data2, q);
```

SYCL_PRINT_EXECUTION_GRAPH
tinyurl.com/dag-print

# USM - Data Dependency in Queues

A more simplified way of specifying dependency as parameter of parallel_for




```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.parallel_for <class taskA>(range<1>(N), [=](id<1> i){
  data1[i] += 2;
});
auto e2 = q.parallel_for <class taskB>(range<1>(N), [=](id<1> i){
  data2[i] += 3;
});
q.parallel_for <class taskC>(range<1>(N), {e1, e2}, [=](id<1> i){
  data1[i] += data2[i];
}).wait();

for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data1, q); free(data2, q);
```

# Hands-On Exercise 6(10 minutes)

1) Use an editor to create code example(ex6.cpp)

2) Compile code

```
icpx -fsycl ex6.cpp -o ex6
```

3) Use the SYCL_PI_TRACE to confirm what accelerator the code ran on.

```
SYCL_PI_TRACE=1 ./ex6
```

4)Create dependency graph and convert to png

```
SYCL_PRINT_EXECUTION_GRAPH=always ./ex6

dot -Tpng <filename> -o <outname>
```

This only works if graphviz is installed

5) Edit the code so the dependencies are implemented another way.

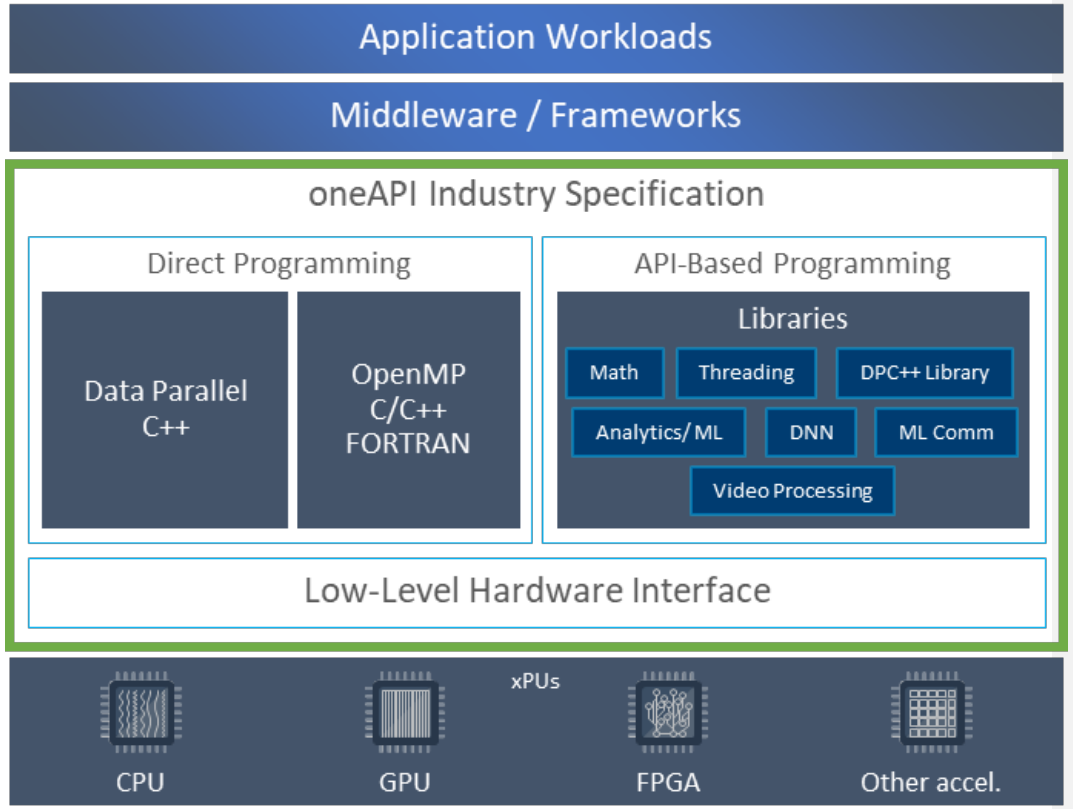
```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main(int argc, char const *argv[])
{
    queue q;

    // Task A
    auto eA = q.submit([&](handler &h)
                       {
        sycl::stream out(1024, 256, h);
        h.parallel_for(N, [=](id<1>)
                       { out << "A"; }); });
    eA.wait();

    // Task B
    auto eB = q.submit([&](handler &h)
                       {
        sycl::stream out(1024, 256, h);
        h.parallel_for(N, [=](id<1>)
                       { out << "B"; }); });

    // Task C
    auto eC = q.submit([&](handler &h)
                       {
        sycl::stream out(1024, 256, h);
        h.depends_on(eB);
        h.parallel_for(N, [=](id<1>)
                       { out << "C"; }); });

    // Task D
    auto eD = q.submit([&](handler &h)
                       {
        sycl::stream out(1024, 256, h);
        h.depends_on({eB,eC});
        h.parallel_for(N, [=](id<1>)
                       { out << "D"; }); });

    return 0;
}
```

# Programmers' perspective: Three things to consider

1. Offload the code to the device

2. Manage the transfer of Data

3. Implement Parallelism



# Types of Kernels

- Basic Kernels
- ND-Range Kernels
- (Heirarchical Kernels)

# Basic Parallel Kernels

## The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

You can also just pass in just a number rather than a range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
            // CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(range<1>(1024), [=](item<1> item){
            id idx = item.get_id();
            range R = item.get_range();
            // CODE THAT RUNS ON DEVICE
});
```

# Using number instead of Range

The `parallel_for` overload without an offset can be called with either a number or a `braced-init-list` with 1-3 elements. In that case the following calls are equivalent:

- `parallel_for(N, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for({N}, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for({N1, N2}, some_kernel)` has same effect as `parallel_for(range<2>(N1, N2), some_kernel)`
- `parallel_for({N1, N2, N3}, some_kernel)` has same effect as `parallel_for(range<3>(N1, N2, N3), some_kernel)`

Below is an example of invoking `parallel_for` with a number instead of an explicit `range` object.

Chapter 4. SYCL programming interface | 301

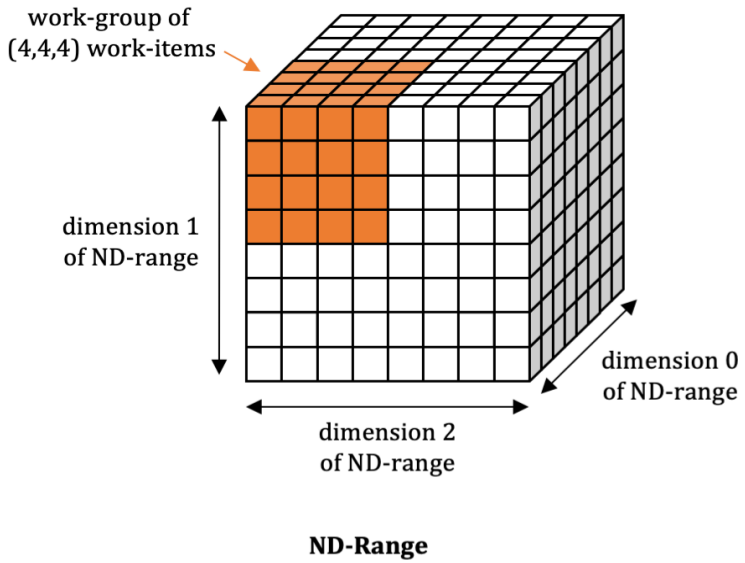
https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

# Basic Kernel (USM)

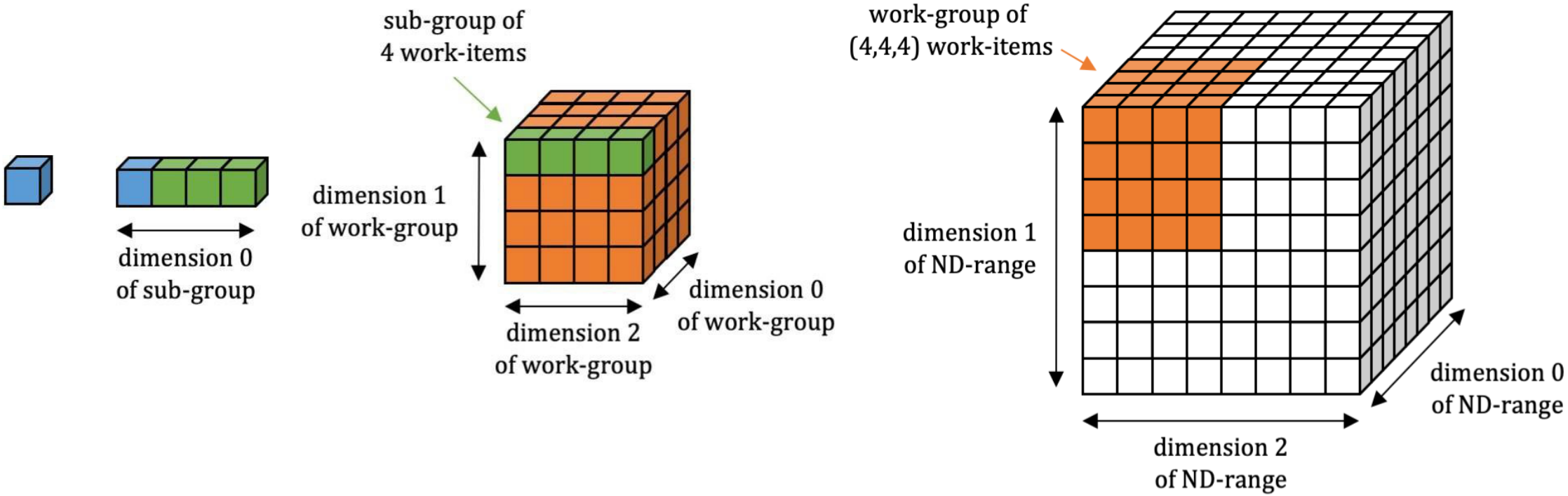
```
int main() {
  const int N = 1000;

  // Create a queue for SYCL execution using the updated selector
  queue q{default_selector_v};
  // Allocate USM shared memory for input and output arrays
  int *a = malloc_shared<int>(N, q);
  int *b = malloc_shared<int>(N, q);
  int *result = malloc_shared<int>(N, q);

  // Initialize input arrays
  for (int i = 0; i < N; i++) {
      a[i] = i;
      b[i] = i;
  }

  // Submit a kernel to add the two arrays
  q.submit([&](handler &h) {
      h.parallel_for(N, [=](id<1> i) {
          result[i] = a[i] + b[i];
      });
  }).wait();

  // Output the result of the first 10 elements
  for (int i = 0; i < 10; i++) {
      std::cout << "result[" << i << "] = " << result[i] << std::endl;
  }
// Free the USM memory
  free(a, q);
  free(b, q);
  free(result, q);

  return 0;
}
```

# ND-range Kernels

- ND-range kernel is another way to express parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.
  - The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
  - The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.



# SYCL Thread Hierarchy and Mapping

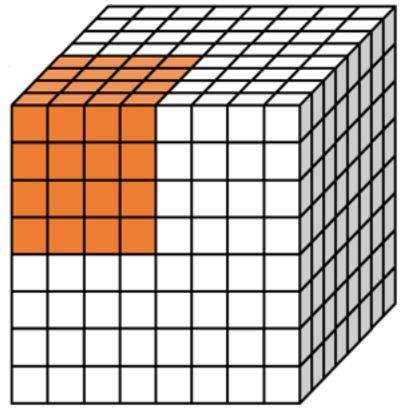



**Work-item** **Sub-group** **Work-group** **ND-Range**

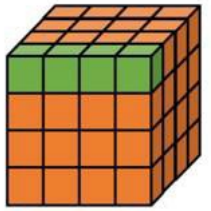
*Thread* *Warp* *Block* *Grid* *(Equivalent CUDA terms)*

# SYCL Thread Hierarchy and Mapping


All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory

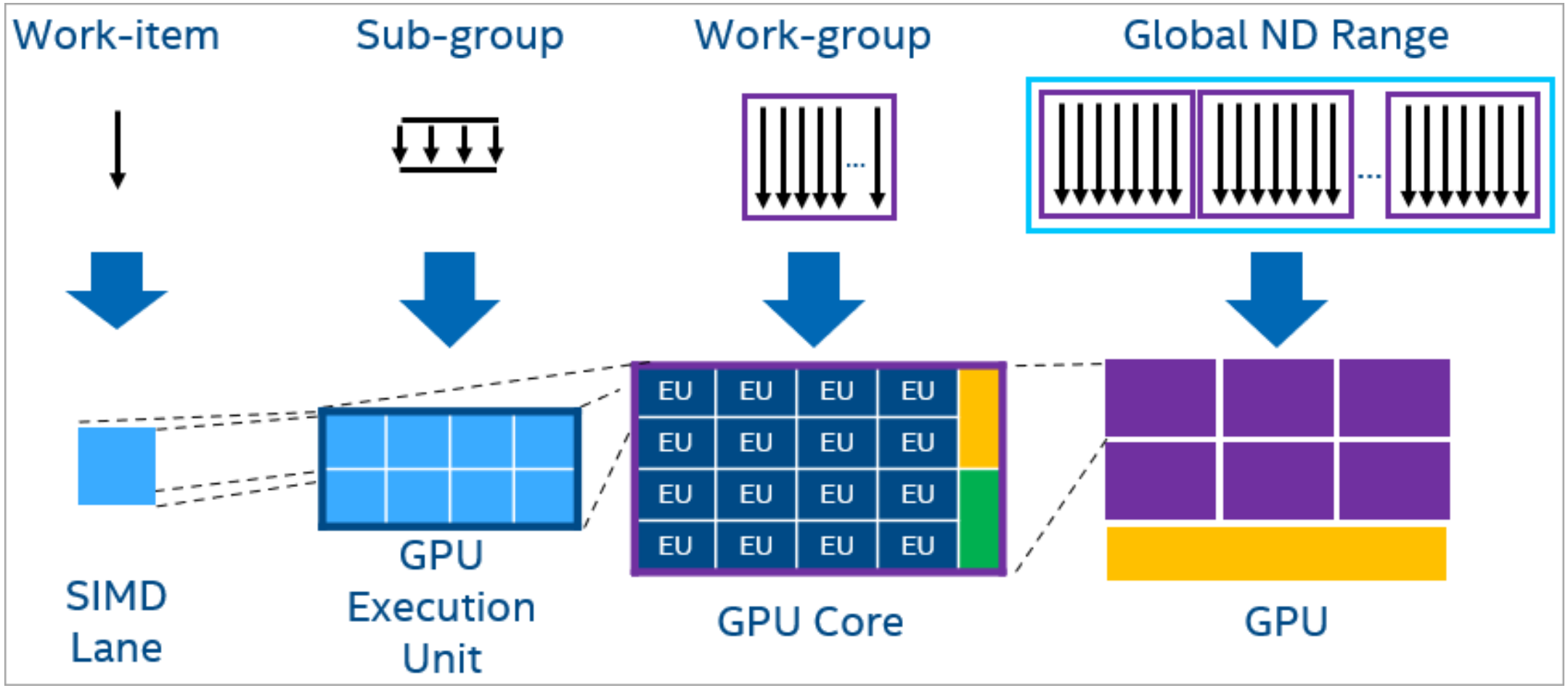


All work-items in a **sub-group** are mapped to vector hardware



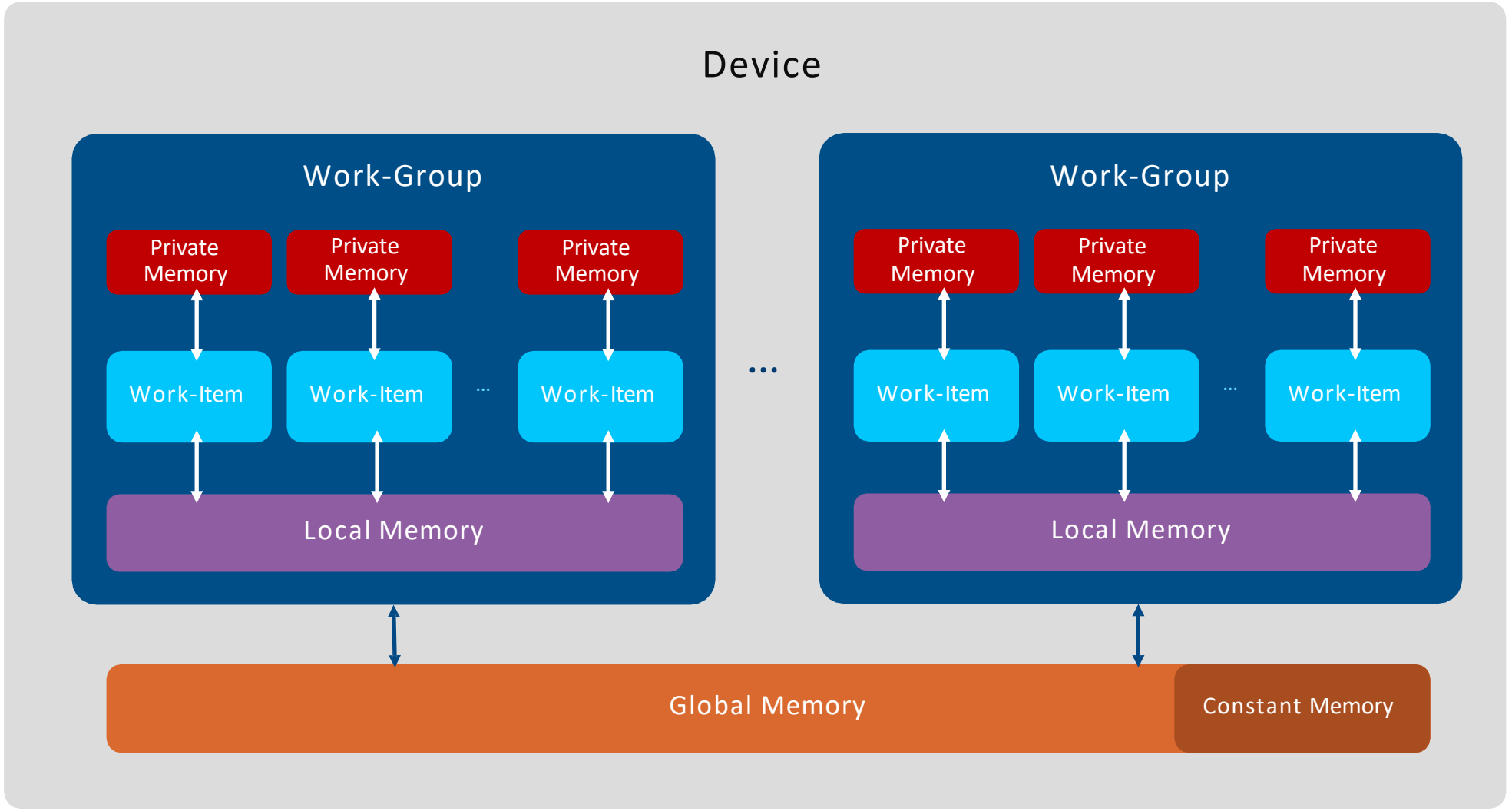
[oneAPI GPU Optimization Guide (intel.com)]

# New Terminology




https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-2/gpu-offload-flow.html

# Logical Memory Hierarchy



# ND-range Kernels

## The functionality of nd_range kernels is exposed via nd_range and nd_item classes

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```

global size

work-group size

nd_range class represents a grouped execution range using global execution range and the local execution range of each work-group.

nd_item class represents an individual instance of a kernel function and allows to query for work-group range and index.

# nd_range Kernel (USM)

```
int main() {
  const int N = 1024;

  // Create a queue for SYCL execution using the updated selector
  queue q{default_selector_v};
  // Allocate USM shared memory for input and output arrays
  int *a = malloc_shared<int>(N, q);
  int *b = malloc_shared<int>(N, q);
  int *result = malloc_shared<int>(N, q);

  // Initialize input arrays
  for (int i = 0; i < N; i++) {
      a[i] = i;
      b[i] = i*2;
  }
```

```
    // Define only the global range
    range<1> global_range(N);

    // Use nd_range but let SYCL determine the best work-group size automatically
    nd_range<1> execution_range(global_range);

    // Submit a kernel using nd_range (SYCL will choose the work-group size)
    q.submit([&](handler &h) {
        h.parallel_for(execution_range, [=](nd_item<1> item) {
            int i = item.get_global_id(0);
            result[i] = a[i] + b[i];
        });
    }).wait();
```

```
  // Output the result of the first 10 elements
  for (int i = 0; i < 10; i++) { std::cout << "result[" << i << "] = " << result[i] << std::endl;}
  // Free the USM memory
    free(a, q);free(b, q);free(result, q);
    return 0;
}
```

# nd_range & work_group_size (USM)

```
int main() {
  const int N = 1024;
  const int work_group_size = 64;

  // Create a queue for SYCL execution using the updated selector
  queue q{default_selector_v};
  // Allocate USM shared memory for input and output arrays
  int *a = malloc_shared<int>(N, q);
  int *b = malloc_shared<int>(N, q);
  int *result = malloc_shared<int>(N, q);

  // Initialize input arrays
  for (int i = 0; i < N; i++) {
      a[i] = i;
      b[i] = i*2;
  }
     // Define the nd_range object separately
     range<1> global_range(N);
     range<1> local_range(work_group_size);
     nd_range<1> execution_range(global_range, local_range);

     // Submit a kernel using nd_range
     q.submit([&](handler &h) {
         h.parallel_for(execution_range, [=](nd_item<1> item) {
             int i = item.get_global_id(0);
             result[i] = a[i] + b[i];
         });
     }).wait();

  // Output the result of the first 10 elements
  for (int i = 0; i < 10; i++) { std::cout << "result[" << i << "] = " << result[i] << std::endl;}
  // Free the USM memory
    free(a, q);free(b, q);free(result, q);
    return 0;
}
```

# When should I specify work-group size?

- **SYCL Behavior with Work-Group Sizes:**
- If you **don't specify** a work-group size, SYCL will internally choose a size that is compatible with the device.
- If you **do specify** a work-group size, it must be compatible with the device's limits (e.g., max work-group size) and the problem size should ideally be divisible by the work-group size for best efficiency.
- **Summary:**
- **For general-purpose and portable code**: It's common not to specify a work-group size and let SYCL handle it.
- **For performance-critical, device-tuned code**: Specifying the work-group size is useful for fine-tuning performance, particularly for GPUs.

# Error Handling

# Error Handling

## Synchronous exceptions

- Detected immediately
  - Failure to construct an object, e.g., can't create a buffer
- Use try...catch block

```
try {
  device_queue.reset(new queue(device_selector));
}
catch (exception const& e) {
  std::cout << "Caught a synchronous SYCL
exception:" << e.what();
  return;
}
```

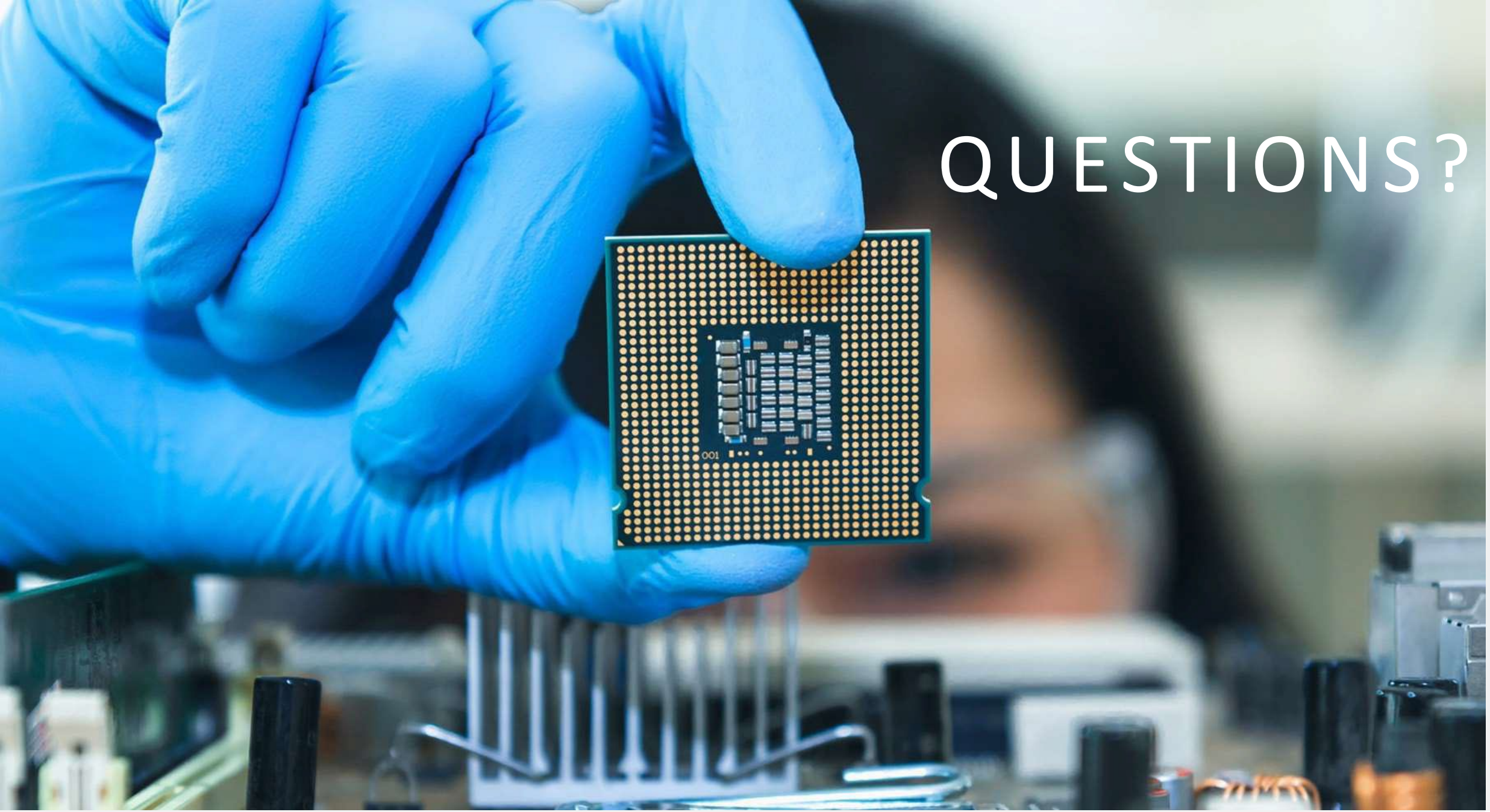
## Asynchronous exceptions

- Caused by a future failure
- Detected immediately
  - E.g., error occurring during execution of a kernel on a device
  - Host program has already moved on to new things!
- Programmer provides processing function, and says when to process

```
auto async_exception_handler = [](exception_list
exceptions) {
  for (std::exception_ptr const& e : exceptions) {
    try {
      std::rethrow_exception(e);
    }
    catch (exception const& e) {
      std::cout << "Caught the Asynchronous SYCL
exception"
        << e.what() << std::endl;
    }
  }
};
```

- queue::wait_and_throw(), queue::throw_asynchronous(), event::wait_and_throw()


intel


QUESTIONS?

# Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.